VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Paralelní implementace ortogonalizace matice

Parallel implementation of matrix orthogonalization

2013 Radim Sojka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Zadání bakalářské práce

Student: **Radim Sojka**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 1103R031 Výpočetní matematika

Téma: Paralelní implementace ortogonalizace matice
Parallel implementation of matrix orthogonalization

Zásady pro vypracování:

V mnoha inženýrských aplikacích se setkáváme s problémem vyjádření vektoru z vektorového prostoru v ortogonální bázi, projekce vektoru do ortogonálních podprostorů apod. Je to právě ortogonalita, která přispívá k významné redukci výpočetní náročnosti a práce s ortogonální bází je vždy příjemnější než s tou neortogonální. Rostoucí výkon paralelních počítačů umožňuje řešení větších a větších úloh, ale s tím se vynořuje celá řada nových problémů žádajících si nová řešení, nové přístupy. Jeden z těchto problémů souvisí s ortogonalizací matic. Je zajímavé, že takový nástroj pro paralelizaci vědeckých výpočtů jako je PETSc, neobsahuje žádnou funkci pro tuto významnou operaci.
Student by se měl seznámit s technikami výpočtu ortogonální báze, tj. ortogonalizací sloupců matice jako je klasický, modifikovaný, iterační Gram-Schmidtův proces, Householderovy transformace, Givensovy rotace, inverze Choleského faktorizace normálové matice nebo jiné maticové rozklady jako QR, SVD apod. Součástí práce by měla být analýza vhodnosti pro paralelizaci (výpočetní náročnost, množství komunikace, paměťové nároky), stability výpočtu (zaokrouhlovací chyby) a porovnání paralelní škálovatelnosti jednotlivých PETSc implementací.
In many engineering applications we have to express a vector from vector space in orthogonal basis, or to project vector into orthogonal subspaces. The orthogonality is this nice feature reducing computation complexity and the work with orthogonal basis is always nicer in comparison to work with non-orthogonal basis. Incresing performance of parallel computers enables the solution of larger and larger problems, but new problems arise and require new solutions and new approaches. One of them deals with matrix orthogonalization. It is interesting that such a tool for the parallelization of scientific computations like PETSc does not contain any function for this important action yet.
Student should get know various techniques of orthogonal basis computation, i.e. matrix columns' orthogonalization, as the classical, modified, iterative Gram-Schmidt process, Householder transformations, Givens rotations, the inverse of Cholesky factor of normal matrix or other matrix factorizations like QR, SVD etc. The work should contain also the suitability analysis for the parallelization (computational complexity, communication amount, memory requirements), stability discussion (rounding errors) and the comparison of parallel scalability of various PETSc implementations.

Seznam doporučené odborné literatury:

[1] Golub, Gene H.; Van Loan, Charles F. (1996), Matrix Computations (3rd ed.), Johns Hopkins, ISBN 978-0-8018-5414-9 .
[2] Lingen, F. J.; Efficient Gram-Schmidt orthonormalization on parallel computers, Research report, Department of Aerospace Engineering, Delft University of Technology, 1999.
[3] http://www.mcs.anl.gov/petsc/

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. David Horák, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. RNDr. Jiří Bouchala, Ph.D.
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 3. května 2013 .....

Chtěl bych poděkovat vedoucímu této bakalářské práce, Ing. Davidu Horákovi, Ph.D., za pomoc, cenné rady a množství času, který mi věnoval.

Abstrakt

Tato bakalářská práce se zabývá paralelní implementací ortogonalizace matice. Ortogonalita je využívána v mnohých aplikacích a může pomoci k usnadnění výpočtu složitých inženýrských úloh. Práce nejprve popisuje některé způsoby ortogonalizace matice a následně se pak zabývá paralelizací různých verzí algoritmů Gramova-Schmidtova ortogonalizačního procesu. Je zde popsán způsob implementace pomocí knihovny PETSc, využívané pro paralelizaci vědeckých výpočtů. Na numerických experimentech je pak srovnána paralelní škálovatelnost a stabilita jednotlivých algoritmů.

Klíčová slova: Ortogonalizace, ortogonální matice, paralelizace, paralelní škálovatelnost, Gramův-Schmidtův proces, PETSc, SLEPc

Abstract

This bachelor thesis deals with parallel implementation of matrix orthogonalization. The orthogonality is used in many applications and can help in engineering calculations. This work describes some orthogonalization methods first, then it focuses on parallelization of the variants of the Gram-Schmidt orthogonalization process. The implementation using PETSc library is described here. PETSc is the tool for the parallelization of the scientific computations. The paralel scalability and the stability of the algorithms are compared on numerical experiments.

Keywords: Orthogonalization, orthogonal matrix, parallelization, parallel scalability, Gram–Schmidt process, PETSc, SLEPc

Seznam použitých zkratek a symbolů

CGS	–	Klasický Gramův-Schmidtův algoritmus
FETI	–	Finite element tearing and interconnect
ICGS	–	Iterovaný klasický Gramův-Schmidtův algoritmus
MGS	–	Modifikovaný Gramův-Schmidtův algoritmus
MPI	–	Message Passing Interface
QP	–	Quadratic programming
$\mathbb{R}$	–	Množina všech reálných čísel
SVD	–	Singular value decomposition
TFETI	–	Total finite element tearing and interconnect

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V současné době, kdy roste výkon a probíhá velký rozvoj paralelních počítačů, je možné řešit náročnější inženýrské úlohy. V těch se často můžeme setkat s problémem ortogonality vektorů. Ortogonalita pak často poskytuje mnoho výhod pro další výpočet. Pomáhá například při snížení vypočetní náročnosti daného problému. Velký význam má ortogonalizace například při výpočtu vlastních čísel matice. Jedním z možných využití je také v metodách rozložení oblastí typu FETI, kde ortogonalita může usnadnit výpočet [1]. Na maticích z jednoho takového problému budou také uvedené ortogonalizační algoritmy testovány.

Nyní, kdy se pro rozsáhlé úlohy používají výkonné paralelní počítače, je výhodné zabývat se paralelizací algoritmů pro ortogonalizaci.

Tato práce se zabývá právě paralelizací problému ortogonalizace matice. V práci se budu zabývat paralelizací klasických technik pro ortogonalizaci matic jako jsou různé modifikace Gramova-Schmidtova algoritmu.

Součástí práce je vytvořit pro tyto algoritmy implementace s pomocí knihovny PETSc. Tento nástroj pro programování paralelních vědeckých výpočtů totiž dosud neobsahuje žádnou funkci pro ortogonalizaci.

V numerických experimentech budou poté jednotlivé algoritmy otestovány z hlediska jejich výpočetní náročnosti, množství komunikace, stability výpočtu atd.

2 Ortogonalita matice

Definice 2.1 *Čtvercová matice Q, pro kterou platí*

$$Q^{-1} = Q^T$$

se nazývá ortogonální matice.

Pro tuto matici pak také platí

$$Q^T Q = I = Q Q^T.$$

Sloupce a také řádky této matice tvoří ortonormální množinu vektorů.

Poznámka 2.1 Často se zabýváme problémem ortogonalizace obdélníkové matice $A \in \mathbb{R}^{m,n}$, kde $m > n$. Výsledkem ortogonalizace je pak matice Q stejné dimenze, jejíž sloupce jsou ortonormální a platí $Q^T Q = I$.

Následující kapitoly se věnují některým maticovým rozkladům souvisejicích s ortogonalitou a metodám ortogonalizace.

2.1 Maticové rozklady

2.1.1 QR rozklad

Věta 2.1 *Nechť $A \in \mathbb{R}^{m,n}$, kde $m > n$, je libovolná matice. Pak existují matice $Q \in \mathbb{R}^{m,n}$ a $R \in \mathbb{R}^{n,n}$ takové, že sloupce matice Q jsou ortonormální a R je horní trojúhelníková matice a navíc platí*

$$A = QR.$$

Důkaz. Existence QR rozkladu vyplývá ze způsobu jeho konstrukce. Pro výpočet QR rozkladu lze použít metody popsané v následujících kapitolách. ∎

2.1.2 Singulární rozklad

Věta 2.2 *Nechť je dána matice $A \in \mathbb{R}^{m,n}$. Pak existují matice $U \in \mathbb{R}^{m,m}$ a $V \in \mathbb{R}^{n,n}$, které jsou ortogonální a diagonální matice $\Sigma \in \mathbb{R}^{m,n}$, pro které platí*

$$A = U \Sigma V^T.$$

Pro diagonální prvky matice Σ navíc platí $\sigma_1 > \sigma_2 > \ldots > \sigma_{min(m,n)} > 0$ a nazývají se singulární čísla matice A. Součin $A = U \Sigma V^T$ nazýváme singulárním rozkladem (SVD) matice A.

Důkaz. Důkaz věty lze nalézt v [6]. ∎

Singulární rozklad lze využít pro mnoho aplikací jako např. řešení soustavy rovnic se singulární maticí, analýza rozptylu ve statistice, filtrování signálů, komprese dat aj. Výpočet singulárního rozkladu má nevýhodu vysoké výpočetní náročnosti [6].

Pro zefektivnění výpočtu SVD se nabízí paralelizace, která je ovšem pro tento problém nesnadná. Možný přístup k zefektivnění výpočtu pomocí paralelizace je ukázán v [6].

2.2 Gramův-Schmidtův ortonormalizační proces

Gramův-Schmidtův proces umožňuje pro zadané vektory $a_1, a_2, \dots a_n \in \mathbb{R}^m$, kde $m > n$ sestavit ortonormální bázi $q_1, q_2, \dots q_n \in \mathbb{R}^m$ podprostoru generovaného zadanými vektory.

Proces probíhá iteračně, kdy v k-té iteraci je vypočten jeden vektor q_k. Nejprve provedeme volbu

$$q_1 = \frac{a_1}{\|a_1\|}.$$

V k-tém kroce, kdy známe vektory $q_1 \dots q_{k-1}$, můžeme rozepsat vektor a_k jako lineární kombinaci vektorů $q_1, \dots q_k$.

$$a_k = \alpha_1 q_1 + \alpha_2 q_2 + \dots + \alpha_k q_k. \tag{2.1}$$

Označme $v = \alpha_k q_k$. Předchozí rovnici můžeme přepsat do tvaru

$$v_k = a_k - \alpha_1 q_1 - \alpha_2 q_2 - \dots - \alpha_{k-1} q_{k-1}. \tag{2.2}$$

Požadujeme, aby vektor v_k byl ortogonální k již vypočteným vektorům $q_1 \dots q_{k-1}$. Proto po přenásobení vektorem $q_i, i \in 1, 2, \dots k-1$ dostáváme

$$0 = (v_k, q_i) = (a_k, q_i) - \alpha_1 (q_1, q_i) - \dots - \alpha_i (q_i, q_i) - \dots - \alpha_k (q_k, q_i). \tag{2.3}$$

Z rovnice 2.3 můžeme vypočítat $\alpha_i = (a_k, q_i)$. Po výpočtu všech skalárů α dostáváme vektor v_k a jeho normalizací získáme výsledný vektor

$$q_k = \frac{v_k}{\|v_k\|}.$$

Algoritmicky v každé iteraci cyklu probíhá výpočet $q_k = f(Q_k, a_k)$, kde sloupce matice Q_k tvoří vektory $q_1, \dots q_{k-1}$. Pro výpočet ortonormální báze existuje několik různých způsobů implementace funkce f a tedy různých verzí Gramova-Schmidtova procesu. Jejich implementace se liší výslednou přesností, rychlostí či vhodností k paralelizaci. Následující kapitoly se zabývají právě třemi různými algoritmy Gramova-Schmidtova procesu.

2.2.1 Modifikovaný Gramův-Schmidtův algoritmus

Jedním z často užívaných algoritmů je modifikovaný Gramův-Schmidtův algoritmus (MGS). Matlabovská implementace algoritmu je uvedena ve výpisu 1.

Při ortogonalizaci vektoru a_k k již ortogonálním vektorům $q_1, \ldots, q_{k-1}$ probíhá cyklus, v jehož každé iteraci je nejprve vypočten prvek s_i jako skalární součin vektoru q_i a vektoru q_k^{i-1}, který dostaneme z původního vektoru a_k po předchozích $i-1$ iteracích. Vypočtený skalár s_i je v této iteraci ihned použit pro výpočet vektoru q_k^i, odečtením násobku vektoru q_i od předešlého vektoru a_k.

V algoritmu MGS tedy není vnitřní skalární součin počítán vždy s původním vektorem a_k, ale s již pozměněným vektorem q_k^{i-1} z předešlých iterací. Díky tomu je tento algoritmus numericky stabilnější než následující algoritmus CGS [2].

```
function [q_k,s] = mgs(Q_k, a_k)

    k = size(Q_k,2);
    s = zeros(k+1,1);
      q_k = a_k;

    for i = 1:k
        s(i) = Q_k(:,i)'*a_k;
        q_k = q_k - s(i)*Q_k(:,i);
    end

    s(k+1) = norm(q_k);
    q_k = (q_k/s(k+1));
end
```

Výpis 1: Algoritmus MGS v Matlabu

2.2.2 Klasický Gramův-Schmidtův algoritmus

Dalším používaným algoritmem je klasický Gramův-Schmidtův algoritmus (CGS), který je velmi podobný předchozímu algoritmu. Výpis 2 zobrazuje jeho matlabovskou implementaci.

Narozdíl od předchozího MGS jsou zde všechny prvky vektoru s počítány z původního vektoru a_k. Až po výpočtu všech skalárních součinů je v následujícím cyklu, po odečtení násobků vektorů $q_1 \ldots q_{k-1}$, získán ortogonální vektor. Uvedený kód je ekvivaletní s výpočtem:

$$\begin{aligned} s &= Q_k^T a_k, \\ q_k &= a_k - Q_k s, \\ q_k &= \frac{q_k}{\|q_k\|}. \end{aligned}$$

Tato verze Gramova-Schmidtova procesu je výhodnější pro paralelní implementaci, protože při ní probíhá podstatně méně komunikace mezi procesy v porovnání s MGS [2].

```
function [q_k,s] = cgs(Q_k, a_k)

    k = size(Q_k,2);
    s = zeros(k+1,1);
    q_k = a_k;

    for i = 1:k
        s(i) = Q_k(:,i)'*a_k;
    end
    for j = 1:k
        q_k = q_k - s(j)*Q_k(:,j);
    end

    s(k+1) = norm(q_k);
    q_k = q_k/s(k+1);
end
```

Výpis 2: Algoritmus CGS v Matlabu

2.2.3 Iterovaný klasický Gramův-Schmidtův algoritmus

Zvláště při výpočtech s většími maticemi může dojít při aplikování CGS algoritmu ke ztrátě ortogonality výsledné matice. Řešením může být použití iteračního zpřesnění.

Výpis 3 ukazuje iterovanou variantu klasického Gramova-Schmidtova algoritmu (ICGS). Vnitřní struktura CGS je zachována. Důležitou částí algoritmu je podmínka

$$\left\| q_k^l \right\| > \alpha \left\| q_k^{l-1} \right\|, \tag{2.4}$$

kde l značí pořadí iterace cyklu while. Při nesplnění (2.4) dochází k další iteraci, během které je znovu ortogonalizován právě vypočtený vektor q_k^l, a tím dochází ke zpřesnění výpočtu.

Důležitým faktorem pro přesnost a rychlost tohoto algoritmu je volba hodnoty α. Vhodnou hodnotu se pokoušel najít Walter Hoffmann při testování množiny vektorů matice typu 210 x 100. Přišel na to, že při volbě $\alpha \approx 0,5$ jsou algoritmem ICGS generovány ortonormální vektory s dostatečnou přesností i pro matice s číslem podmíněnosti řádu 10^{10}. Navíc pro tuto hodnotu algoritmus ICGS konvergoval vždy během dvou iterací ve všech jeho numerických experimentech [2].

ICGS je stejně jako CGS výhodnější pro paralelní implementaci v porovnání s algoritmem MGS, protože zde dochází k menšímu množství komunikace mezi procesory. Proto může být ve výsledku rychlejší než MGS, přestože při něm probíhá více výpočetních operací [2].

```
function [q_k,s] = icgs(Q_k, a_k)

    k = size(Q_k,2);
    s = zeros(k+1,1);
    q_k = a_k;
    pom_q = a_k;
    pom_s = s;
    con = true;
    alfa = 0.5;

    if k > 0
        while con
            for i = 1:k
                pom_s(i) = Q_k(:, i)'*q_k;
            end
            for j = 1:k
                pom_q = pom_q - pom_s(j)*Q_k(:,j);
            end
            s = s + pom_s;
            if norm(pom_q) > alfa * norm(q_k)
                con = false;
            end
            q_k = pom_q;
        end
    end

    s(k+1) = norm(q_k);
    q_k = q_k/s(k+1);
end
```

Výpis 3: Algoritmus ICGS v Matlabu

2.3 Givensova transformace

Givensova metoda pro ortogonalizaci využívá matice rovinné rotace (Givensova matice). Její odvození lze ukázat na příkladě otočení bodu v rovině kolem počátku [3].

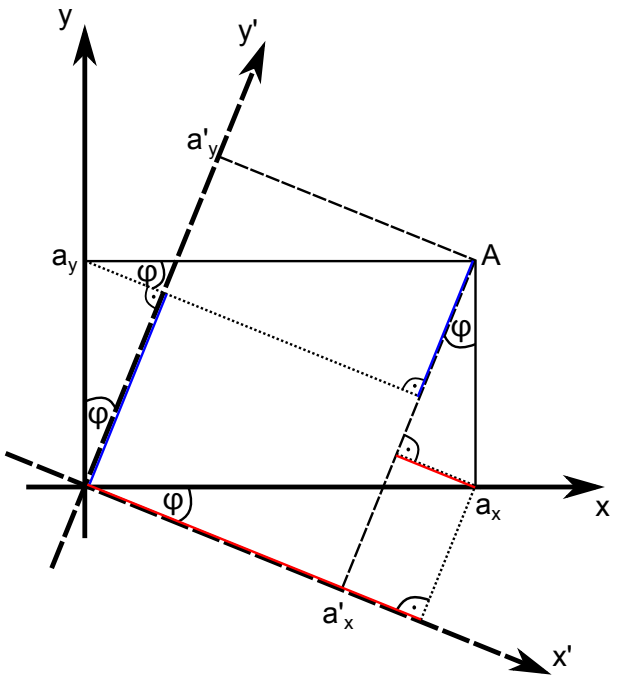
Mějme bod $A = [a_x, a_y]$, který chceme otočit o úhel φ. Touto operací získáme bod $A' = [a'_x, a'_y]$. Úlohu můžeme přeformulovat tak, že hledáme souřadnice bodu A, s pootočenými osami souřadnic x', y' o úhel $-\varphi$ viz obr. 1.

Pomocí goniometrických funkcí odvodíme vztahy pro nové souřadnice bodu A

$$a'_x = a_x \cos(\varphi) - a_y \sin(\varphi)$$
$$a'_y = a_x \sin(\varphi) + a_y \cos(\varphi).$$

Získáváme tak matici rovinné rotace, která otočí bod A kolem počátku o úhel φ

$$G = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{bmatrix}.$$



Obrázek 1: Odvození Givensovy matice

Nyní uvažujme prostor $\mathbb{R}^n$, matici G, která bude provádět rotaci v rovině i, j zavedeme

$$G_{i,j}(\varphi) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & \cos(\varphi) & & -\sin(\varphi) & & \\ & & & \ddots & & & \\ & & \sin(\varphi) & & \cos(\varphi) & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix}.$$

Jedná se o jednotkovou matici, kde na odpovídajících pozicích v $g_{i,i}, g_{i,j}, g_{j,i}$ a $g_{j,j}$ jsou vloženy prvky matice rovinné rotace.

Při výpočtu QR rozkladu je tato matice využívána k nulování prvků jednotlivých sloupců původní matice [3]. Mějme vektor x a budeme chtít vynulovat prvek x_j pomocí matice G která provádí rotaci v rovině i, j. Budeme tedy provádět operaci

$$y = G^T x. \tag{2.5}$$

Pro prvky vektoru y požadujeme

$$y_i = \sqrt{x_i^2 + x_j^2}, \quad y_j = 0. \tag{2.6}$$

Hodnoty $\sin(\varphi)$ a $\cos(\varphi)$ tak můžeme získat ze soustavy rovnic vzniklé z 2.5 a 2.6

$$\begin{aligned} x_i \cos(\varphi) + x_j \sin(\varphi) &= \sqrt{x_i^2 + x_j^2} \\ -x_i \sin(\varphi) + x_j \cos(\varphi) &= 0. \end{aligned} \tag{2.7}$$

Dostáváme

$$\sin(\varphi) = \frac{x_j}{\sqrt{x_i^2 + x_j^2}}, \quad \cos(\varphi) = \frac{x_i}{\sqrt{x_i^2 + x_j^2}}. \tag{2.8}$$

QR rozklad matice A provedeme nulováním prvků matice postupně po jednotlivých sloupcích. Matici A převedeme na horní trojúhelníkovou matici R

$$A = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \overset{G_{3,4}^T}{\rightarrow} \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ 0 & * & * & * \end{bmatrix} \overset{G_{2,3}^T}{\rightarrow} \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \overset{G_{1,2}^T}{\rightarrow} \cdots \overset{G_{2,3}^T}{\rightarrow} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix} \overset{G_{3,4}^T}{\rightarrow}$$

$$\overset{G_{3,4}^T}{\rightarrow} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{bmatrix} = R.$$

Zbývá získat matici Q. Označme matice rovinné rotace $G_1, G_2, ..., G_m$ postupně tak jak byly aplikovány na matici A. Pak můžeme psát

$$A = QR \;\Rightarrow\; Q^T A \;=\; R \;=\; G_m^T G_{m-1}^T ... G_1^T A \;\Rightarrow\; Q \;=\; G_1 G_2 ... G_m.$$

Matice $G_1, G_2, ..., G_m$ jsou ortogonální, proto také součin těchto matic je ortogonální.

```
function [ Q, R ] = Givens_trans( A )
    m=size(A,1);
    n=size(A,2);
    Q = eye(m,n);
    R = A;

    for i=1:n
        for j=m:-1:i+1
            x=R(:,i);
            c=x(j-1)/sqrt((x(j-1))^2+(x(j))^2);
            s=x(j)/sqrt((x(j-1))^2+(x(j))^2);
            G = eye(m);
            G(j-1,j-1:j)=[c,-s];
            G(j,j-1:j)=[s,c];
            R=G'*R;
            Q=Q*G;
        end
    end
end
```

Výpis 4: Givensova transformace v Matlabu

2.4 Householderova transformace

Podobně jako v předchozí kapitole je možné pro výpočet QR rozkladu využít Householderovy matice zrcadlení H. Odvození matice lze ukázat na úloze projekce vektoru do osy. Budeme požadovat, abychom přenásobením maticí H získali projekci vektoru r do osy x stejné délky (viz obr. 2) [3].

Pro prostor $\mathbb{R}^2$ má vektor Hr tvar

$$Hr = \begin{bmatrix} \pm \|r\| \\ 0 \end{bmatrix}.$$

Z obrázku je patrné, že vektor Hr je zrcadlovým obrazem podle osy souměrnosti o, která je kolmá na vekotor v, který získáme vztahem

$$v = Hr - r. \tag{2.9}$$

Vektor Hr lze tedy vyjádřit ve tvaru

$$Hr = r + v. \tag{2.10}$$

Vektor v můžeme také rozepsat

$$v = \frac{v}{\|v\|} 2 \|r\| \cos\ \alpha, \tag{2.11}$$

kde $\frac{v}{\|v\|}$ je jednotkový vektor vyjadřující směr a výraz $2 \|r\| \cos\ \alpha$ vyjadřuje velikost vektoru v.

Po rozšíření vztahu normou $\|v\|$ lze využít definice skalárního součinu

$$v^T r = \|v\| \|r\| \cos(\pi - \alpha) = - \|v\| \|r\| \cos\ \alpha. \tag{2.12}$$

Můžeme tedy upravit

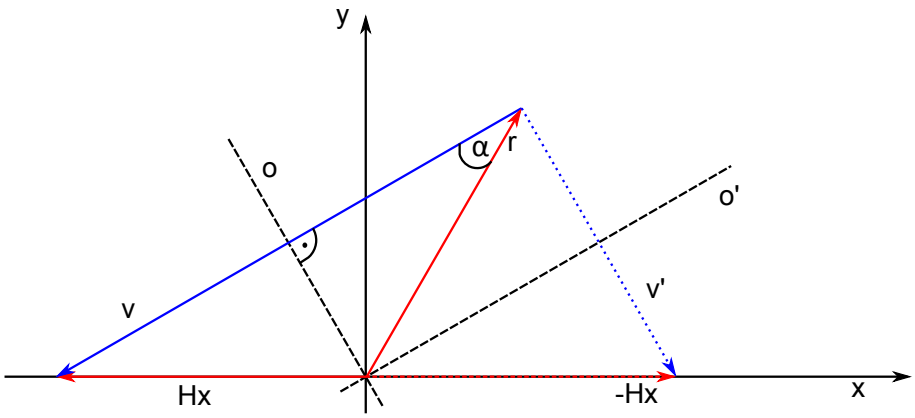
$$v = \frac{v}{\|v\|} 2 \|r\| \cos\ \alpha = \frac{v}{\|v\|^2} 2 \|r\| \cos\ \alpha = -2 \frac{v}{\|v\|^2} v^T r. \tag{2.13}$$

Po dosazení do (2.10) dostáváme

$$Hr = r + v = r - 2 \frac{vv^T}{\|v\|^2} r = \left(I - 2 \frac{vv^T}{\|v\|^2} \right) r. \tag{2.14}$$

Z tohoto vztahu již můžeme vyjádřit vztah pro Householderovu matici

$$H = I - 2 \frac{vv^T}{\|v\|^2}. \tag{2.15}$$



Obrázek 2: Odvození Householderovy matice

Z obrázku 2 je vidět, že vektor Hr není definován jednoznačně. Abychom zmenšili vliv zaokrouhlovacích chyb, požadujeme aby vektor r nebyl příliš blízko svému obrazu Hr [3]. Vektor v tedy volíme

$$v = -sign(r_1)\,\|r\|\,e_1 - r, \tag{2.16}$$

kde $e_1 = (1, 0, \cdots, 0)^T$ je první vektor standardní báze.

Pro výpočet QR rozkladu za pomocí matic zrcadlení postupujeme podobně jako v případě Givensovy transformace. Postupně nulujeme prvky matice A tak, že vznikne horní trojúhelníková matice R

$$A = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \overset{H_1}{\rightarrow} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{bmatrix} \overset{H_2}{\rightarrow} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{bmatrix} \overset{H_3}{\rightarrow} \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{bmatrix} = R.$$

Nejprve sestavíme matici $\tilde{H}_k \in \mathbb{R}^{(n-k+1),(n-k+1)}$, která bude nulovat prvky k-tého sloupce matice $A \in \mathbb{R}^{n,n}$. Vektor r, pomocí kterého vypočteme matici $\tilde{H}_k$, tvoří prvky tohoto sloupce pod diagonálou včetně. Výsledná matice H_k má tvar

$$H_k = \begin{bmatrix} I_k & 0 \\ 0 & \tilde{H}_k \end{bmatrix},$$

kde matice $I_k \in \mathbb{R}^{(k-1),(k-1)}$.

Můžeme tedy psát

$$A = QR \;\Rightarrow\; Q^T A \;=\; R \;=\; H_n H_{n-1} ... H_1 A \;\Rightarrow\; Q \;=\; H_1 H_2 ... H_n.$$

Matice $H_1, H_2, ..., H_n$ jsou symetrické ortogonální matice, neboť

$$H^T H = \left(I - 2\frac{vv^T}{\|v\|^2}\right)^T \left(I - 2\frac{vv^T}{\|v\|^2}\right) = II - \frac{4}{\|v\|^2}vv^T + \frac{4}{\|v\|^4}\left(vv^T vv^T\right) =$$

$$= I - \frac{4}{\|v\|^2}vv^T + \frac{4}{\|v\|^4}\left(v\|v\|^2 v^T\right) = I - \frac{4}{\|v\|^2}vv^T + \frac{4}{\|v\|^2}vv^T = I.$$

Součin těchto matic je tedy také ortogonální matice.

```
function [ Q, R ] = H_trans( A )

    m=size(A,1);
    n=size(A,2);
    Q = eye(m);
    R = A;

    for i=1:n
        x=R(i:m,i);
        v=-sign(x(1))*norm(x)*eye(m-i+1,1)-x;
        if norm(v)>0
            v=v/norm(v);
            H=eye(m);
            Hi=eye(m-i+1)-2*v*v';
            H(i:m,i:m) = Hi;
            R=H*R;
            Q=Q*H;
        end
    end
end
```

Výpis 5: Householderova transformace v Matlabu

3 Paralelizace

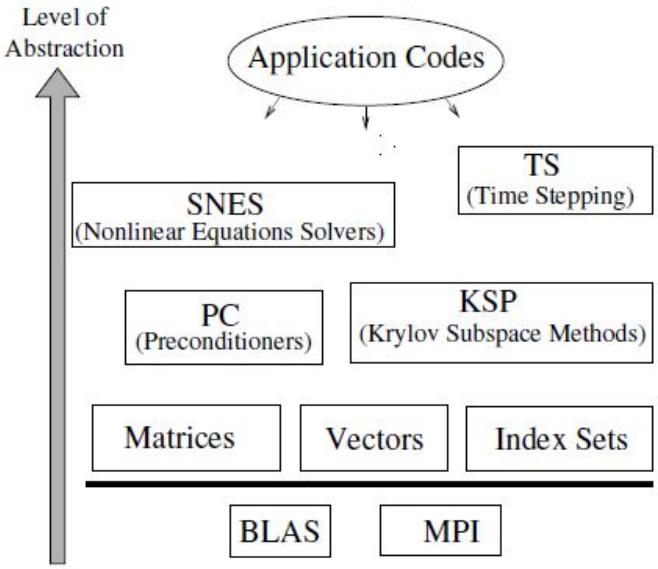
Z algoritmů, které byly popsány v předchozí kapitole, se jeví pro paralelní implementaci jako nejvýhodnější varianty Gramova-Schmidtova procesu. V této kapitole se budu podrobněji věnovat právě paralelizaci těchto algoritmů. Krátce popíšu i nástroj pro implementaci těchto algoritmů, knihovnu PETSc, kterou jsem pro tuto práci používal.

3.1 Knihovna PETSc

Pro implementaci algoritmů byla použita knihovna PETSc 3.3.0 (The Portable, Extensible Toolkit for Scientific Computation). Tato knihovna je vyvíjena skupinou vývojářů z Argonne National Laboratory [4].

PETSc je sada datových struktur a rutin pro paralelní řešení vědeckých aplikací modelovaných podle parciálních diferenciálních rovnic nebo souvisejících problémů na vysoce výkonných počítačích. Knihovnu je možné použít pro programování vědeckých aplikací v jazycích C, C++ nebo Fortranu.

Pro komunikaci mezi výpočetními uzly používá MPI standard. Dále využívá rutiny knihoven BLAS, LAPACK, LINPACK, MINPACK, SPARSPAK a libtfs.



Obrázek 3: Struktura knihovny PETSc [4]

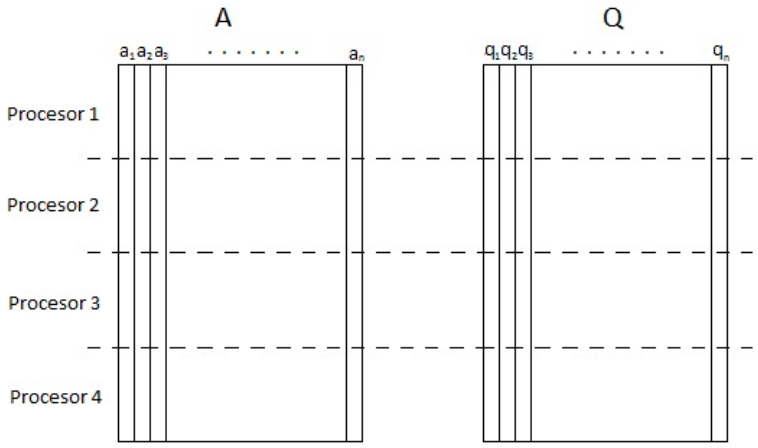
PETSc se skládá z různých knihoven (podobných třídám v C++), které pracují s konkrétní rodinou objektů. Komponenty PETSc jsou množiny indexů, vektrory, matice (husté i řídké), metody Krylova podprostoru, předpodmínění, nelineární řešiče nebo časové krokovače pro výpočet parciálních diferenciálních rovnic závislých na čase.

Obrázek 3 znázorňuje hierarchistické uspořádání knihovny vzhledem k úrovni abstrakce.

3.2 Distribuce vektorů přes procesy

Jestliže chceme ortonormalizovat sloupce vstupní matice A, provedeme distribuci této matice i počítané matice Q na jednotlivé procesy po řádcích. Například distribuce dat pro výpočet se čtyřmi procesy je ukázán na obr. 4.

Díky tomuto rozložení dat na jednotlivé procesy můžeme paralelizovat algoritmy popsané v kapitole 2. Některé operace výpočtu se pak mohou provádět na jednotlivých procesorech nezávisle na sobě bez jakékoliv komunikace mezi procesy (např. odčítání vektorů). Komunikace bude naopak probíhat např. při výpočtu normy vektoru nebo skalárním součinu dvou vektorů.



Obrázek 4: Distribuce dat

Alokaci dat, potřebnou pro uložení matice `A` s výše uvedenou distribucí dat, provedeme pro husté matice PETSc funkcí

```
MatCreateMPIDense(PETSC_COMM_WORLD,PetscInt m,PetscInt n,PetscInt M,
                  PetscInt N,PetscScalar *data,Mat *A),
```

kde `PETSC_COMM_WORLD` je komunikátor zahrnující všechny procesy dostupné pro výpočet, `m` počet lokálních řádků, `n` lokálních sloupců, `M` celkový počet řádků a `N` celkový počet sloupců matice `A`.

Pro vytvoření řídké matice lze podobně použít funkci

```
MatCreateMPIAIJ(PETSC_COMM_WORLD,PetscInt m,PetscInt n,PetscInt M,
                PetscInt N,PetscInt d_nz,const PetscInt d_nnz[],
                PetscInt o_nz,const PetscInt o_nnz[],Mat *A).
```

Parametry `d_nz`, `d_nnz[]`, `o_nz` a `o_nnz[]` specifikují počet nenulových řádků v diagonálních a nediagonálních blocích matice `A`. Význam ostatních parametrů funkce se shoduje s předchozí funkcí.

Označení MPI v názvu funkcí připomíná, že vzniklá matice `A` je paralelní. V případě, že komunikátor obsahuje pouze jeden proces, je možné pro alokaci paměti použít funkci `MatCreateSeqDense()` nebo `MatCreateSeqAIJ()`.

Analogicky jsou vytvořeny také vektory vstupující jako parametry do samotné funkce pro ortonormalizaci vektoru

```
Orthogonalize(Vec a, Vec *q, PetscInt k).
```

Alokace dat pro vektor `a`, který představuje sloupec matice `A`, který bude právě touto funkcí ortonormalizován k množině již ortonormálních vektorů `q`, je provedena funkcí

```
VecCreateMPI(PETSC_COMM_WORLD,PetscInt m,PetscInt M,Vec *a).
```

Vektor `a` musí mít stejné paralelní rozložení jako matice `A`. Hodnotu `m`, která představuje lokální délku vektoru, získáme funkcí

```
MatGetLocalSize(Mat A,PetscInt *m,PetscInt* n).
```

Pak vždy před voláním ortonormalizační funkce vložíme do vektoru `a` hodnoty sloupce matice `A` pomocí další PETSc funkce

```
MatGetColumnVector(Mat A,Vec a,PetscInt col),
```

kde hodnota `col` udává číslo sloupce, který bude v dané iteraci ortonormalizován. Jako poslední je vytvořeno pole vektorů `q`, do kterého se v průběhu výpočtu ukládají již ortonormalizované sloupce původní matice `A`. Jelikož i tyto vektory musí mít stejné paralelní rozložení jako matice `A` (viz obr.4) můžeme využít již vytvořeného vektoru `a` a vytvoříme toto pole vektorů pomocí funkce

```
VecDuplicateVecs(Vec v,PetscInt k,Vec *q[]).
```

3.3 Implementace ortonormalizační funkce

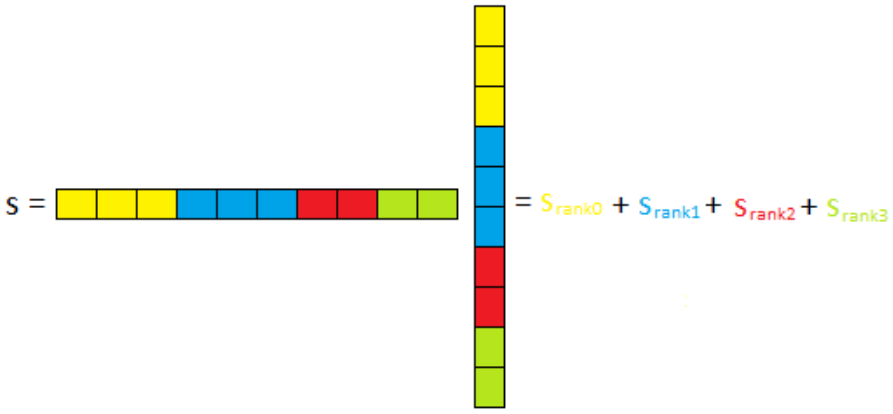
Jádrem výpočtu je funkce, která ortonormalizuje vektor vzhledem k množině již ortonormálních vektorů. Tato funkce se volá sekvenčně pro každý sloupec ortogonalizované matice.

Klíčovou fází funkce je výpočet skalárních součinů právě ortonormalizovaného vektoru s již ortonormálními vektory. Při této operaci dochází ke globální komunikaci (redukci) mezi procesy, kdy jsou sečteny výsledky součinů lokálních vektorů na každém procesoru (viz obr. 5) a výsledný součet poslán všem procesům. Tato operace se při použití MPI standardu vykonává rutinou `MPI_Allreduce`.

K výpočtu skalárního součinu při algoritmu MGS je použita funkce knihovny PETSc

```
VecTDot(Vec x,Vec y,PetscScalar *s),
```

která se při výpočtu rovněž stará o zmíněnou globální komunikaci mezi procesy. Při ortonormalizaci k-tého sloupce je tato funkce zavolána celkem $(k-1)$ krát. Tedy i počet globálních komunikací během výpočtu skalárních součinů je $k-1$.



Obrázek 5: Paralelní násobení vektorů

Algoritmus CGS nepotřebuje oproti MGS pro výpočet takové množství komunikace. Skalární součiny jsou zde vypočteny okamžitě po sobě a jejich výsledek je používán až následně. Díky tomu je možné potřebná data rozeslat pouze jednou globální komunikací současně po vypočtení všech skalárních součinů. To zajistí PETSc funkce

```
VecMDot(Vec x,PetscInt nv,const Vec y[],PetscScalar s[]),
```

kde `x` je vektor pro který je počítán skalární součin se všemi vektory v poli `y` a `nv` je počet vektorů v poli.

Podobně je na tom také algoritmus ICGS. Zde ovšem může proběhnout tato komunikace vícekrát v závislosti na počtu iterací, které jsou potřeba pro dosažení určené přesnosti. V každé iteraci se navíc kvůli vyhodnocení podmínky cyklu musí vypočítat také norma vektoru po ortogonalizaci, při které taktéž probíhá stejná komunikace mezi procesy.

Po skalárních součinech vektorů následuje odečítání dané lineární kombinace vektorů již ortonormálních od ortonormalizovaného vektoru. Toto odečítání lze provést další funkcí knihovny PETSc

```
VecAXPY(Vec y,PetscScalar alpha,Vec x).
```

Ačkoliv k této operaci není potřeba žádné komunikace, při použití této funkce k ní podle profilovacích nástrojů PETSc dochází (viz. příloha A). Jelikož by tento nárůst komunikace mohl zmenšit význam paralelizace, ke každému algoritmu byly vytvořeny dvě implementace. V druhém způsobu implementace je toto odečítání vektorů realizováno pomocí polí reprezentující vektory. Použitím funkce

```
VecGetArray(Vec x,PetscScalar *a[]),
```

každý proces získá pole `a` obsahující lokální část vektoru `x` příslušející danému procesu. Po provedení odčítání vektorů lze data upraveného pole znovu uložit do vektoru `x` funkcí

```
VecRestoreArray(Vec x,PetscScalar *a[]).
```

V příloze A můžeme porovnat množství globální komunikace (redukcí) zmíněných implementací. Při ortogonalizaci matice typu 1680×144 algoritmy, kde se odečítání vektorů realizuje pomocí PETSc funkce, byl počet redukcí při použití MGS přibližně 21000, pro CGS 11000 a pro ICGS 14000. Při použití algoritmů s odečítáním pomocí polí to pak bylo přibližně MGS 10000, CGS 290 a ICGS 520.

Z těchto údajů vidíme, že při použití algoritmu MGS dochází oproti algoritmům CGS a ICGS k podstatně vyššímu množství globální komunikace. Ta může výrazně ovlivnit efektivitu paralelních algoritmů, zvláště pokud jsou komunikační náklady velké v porovnání s výpočetní složitostí. Největší rozdíly v neprospěch MGS lze tedy předpokládat u ortonormalizaci „krátkých“ vektorů.

Výpočetní náročnost algoritmů MGS a CGS je srovnatelná. V porovnání s těmito algoritmy je ICGS výpočetně náročnější v závislosti na počtu zpřesňujících iterací. Počet operací nebo pamětovou náročnost algoritmů lze také nalézt v příloze A.

4 Numerické experimenty

David Horák a Václav Hapla ukázali, že použití ortogonalizace by mohla být jedna z cest pro paralelizaci hrubého problému metody TFETI [1]. Testování algoritmů pro ortogonalizaci bude provedeno na matici G vycházející z aplikace metody TFETI na 2D elastostatickou úlohu ocelového nosníku.

Kromě testování mých PETSc implementací ortogonalizačních algoritmů zde budou pro ortogonalizaci použity také funkce knihovny SLEPc. Výsledky budou vzájemně porovnány.

4.1 TFETI

TFETI (Total Finite Element Tearing and Interconnecting) je metoda rozložení oblastí, která umožňuje rozložit původní rozsáhlou úlohu na menší úlohy, které je možné řešit samostatně a nezávisle na ostatních. Při této metodě se fyzicky oddělují jednotlivé podoblasti. Spojitost na hranicích jednotlivých podoblastí a Dirichletovy okrajové podmínky jsou vynuceny Lagrangeovými multiplikátory [7].

Při aplikaci metody je původní oblast Ω rozdělena na N_s podoblastí Ω^s. Konečně prvková diskretizace s vhodným očíslováním uzlů přechází v QP problém

$$\min \tfrac{1}{2}u^T K u - u^T f \text{ za podmínky } Bu = c,$$

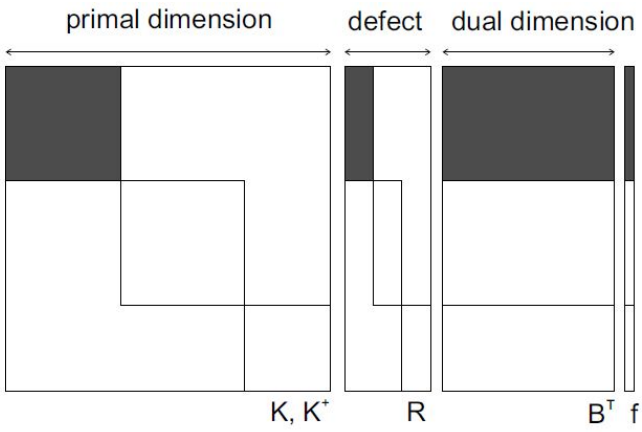
kde

$$K = \begin{bmatrix} K^1 & & \\ & \ddots & \\ & & K^{N_s} \end{bmatrix}, f = \begin{bmatrix} f^1 \\ \vdots \\ f^{N_s} \end{bmatrix}, u = \begin{bmatrix} u^1 \\ \vdots \\ u^{N_s} \end{bmatrix}, B = \begin{bmatrix} B^1 & \cdots & B^{N_s} \end{bmatrix}.$$

K je řídká blokově diagonální matice tuhosti, f sloupcový vektor zatížení, B matice vazebních podmínek, u vektor hledaného posunutí a c vektor vazebních konstant. Matici K tvoří bloky K^s, které jsou maticemi tuhosti jednotlivých oblastí Ω^s. Řádky matice B, s hodnotami -1 a 1 na pozicích příslušných spojovacích uzlů a ostatními prvky nulovými, a vektor c, který má na pozicích, které zajišťují Dirichletovy podmínky, libovolné hodnoty a na všech ostatních nulové prvky, které zajišťují spojitost posunutí na vytvořených hranicích mezi podoblastmi [1].

Pro všechny lokální matice tuhosti K^s jsou předem známé jádra R^s, které mají stejnou dimenzi a mohou být přímo sestavena. Matice R, jejíž sloupce jsou tvořeny bází jádra matice K, má blokově diagonální strukturu

$$R = \begin{bmatrix} R^1 & & \\ & \ddots & \\ & & R^{N_s} \end{bmatrix}.$$



Obrázek 6: Distribuce dat při paralelní implementaci TFETI metody. [1]

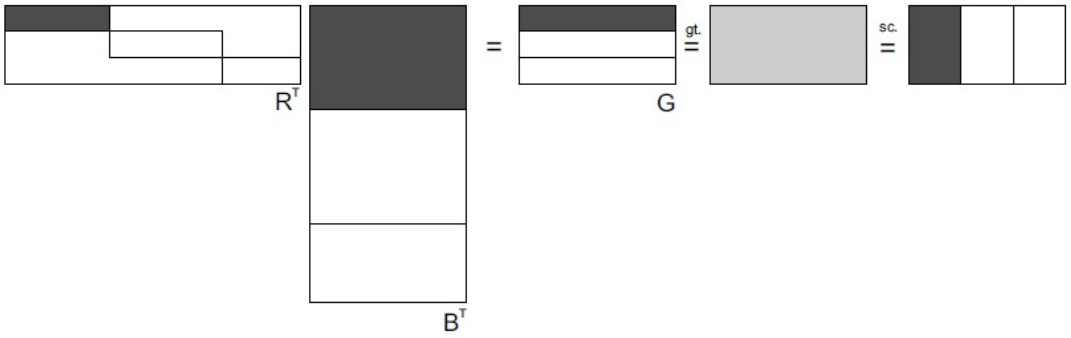
Rozměry vzniklých matic a jejich distribuce při paralelním výpočtu přes jednotlivé procesy je znázorněna na obr. 6.

Využití ortogonalizace přichází při výpočtu ortogonálního projektoru

$$P_G = I - G^T(GG^T)^{-1}G,$$

kde $G = R^T B^T$.

Pokud by matice G měla ortonormální řádky výpočet by se velice zjednodušil, protože $GG^T = I = (GG^T)^{-1}$. Pro efektivní použití ortogonalizačních algoritmů je potřeba matici G nejprve redistribuovat (viz obr. 7).



Obrázek 7: Redistribuce matice G [1].

Obdobně u primárního problému lze využít ortogonalizace matice B pro výpočet projektoru

$$P_B = I - B^T(BB^T)^{-1}B.$$

Numerické experimenty, popsané v této kapitole, probíhaly na matici G získané dekompozicí a diskretizací 2D elastostatické úlohy ocelového nosníku, který je definovaný oblastí $\Omega = (0, 600) \times (0, 200)$, kde rozměry jsou udávány v mm. Vlastnosti materiálu jsou definovány Youngovým modulem $E = 2.1 \cdot 10^5$MPa, Poissonovým číslem $\nu = 0.3$ a hustotou $\rho = 7.85 \cdot 10^{-6}$kg/mm^3. Nosník je upevněn svou kratší stranou v každém směru $\bar{\Gamma}_U = 0 \times [0, 200]$ a zatížen gravitační silou při gravitačním zrychlení $g = 9800$mm/s^2.

4.2 SLEPc

SLEPc (the Scalable Library for Eigenvalue Problem Computations) je softwarová knihovna pro řešení rozsáhlých úloh týkajících se problému vlastních čísel na paralelních počítačích. Je založena na datových strukturách knihovny PETSc a využívá standard MPI pro meziprocesovou komunikaci. Je vyvíjena výzkumníky z Universitat Politécnica de Valencia [5].

Mimo jiné obsahuje knihovna funkce pro ortogonalizaci resp. QR rozklad matice. Implementuje modifikovaný i klasický Gramův-Schmidtův algoritmus včetně jejich iterativních podob.

Pro výpočet ortogonalizace byla při testování využita SLEPc funkce

```
IPQRDecomposition(IP ip,Vec *V,PetscInt m,PetscInt n,PetscScalar *R,
                  PetscInt ldr),
```

která zortonormalizuje pole vektorů `V` (od `m`-tého po `n`-tý vektor) a vrací také matici `R`. Tato funkce využívá pro výpočet QR rozkladu algoritmus CGS s iterativním zpřesněním.

4.3 HECToR

Všechny níže uvedené výsledky byly získány z měření, pro které byl použit superpočítač HECToR (High End Computing Terascale Resources) nacházející se v Edinburghu.

V současnosti je HECToR postaven na systému Cray XE6, který nabízí 704 výpočetních bladů a každý z nich obsahuje čtyři výpočetní uzly. To dává celkem 2816 výpočetních uzlů. Všechny jsou vybavené dvěma 16-jádrovými AMD Opteron 2,3GHz Interlagos procesory. Každý z těchto procesorů má k dispozici paměť o velikosti 16GB. Celkem HECToR obsahuje 90112 jader a k dispozici je kolem 90TB paměti. Jeho teoretický maximální výkon (peak performance) je 800Tflops. Pro komunikaci mezi výpočetními uzly je používán chip Cray Gemini, který má 10 síťových linek použitých k vytvoření sítě ve tvaru 3D anuloidu [8].

4.4 Výsledky měření

Paralelní škálovatelnost byla testována na maticích různých rozměrů. Matice byly generovány pro úlohu, která je popsána v kapitole 4.1. Rozměry této matice byly měněny požadovaným počtem podoblastí a požadovanou diskretizací úlohy.

V tabulce 1 můžeme vidět časy pro ortogonalizaci matice G při rozložení úlohy na 300 podoblastí. Probíhala tak ortonormalizace 900 vektorů délky 22800.

Čas označen jako preprocessing zahrnuje především čas potřebný k vytvoření ortonormalizovaných vektorů výběrem všech sloupců matice. V rámci postprocessingu jsou pak ortonormální vektory ukládány zpět do matice. Samotná ortonormalizace vektorů pak s maticemi nepracuje, a proto jsou časy jednotlivých algoritmů pro řídké i husté matice téměř stejné. V tabulce jsou proto uvedeny pouze časy naměřené při ortogonalizaci řídké matice. Velký rozdíl ovšem můžeme vidět v čase preprocessingu, kde zvláště u hustých matic je výběr vektorů velice pomalý a tato operace často probíhá déle než samotná ortonormalizace vektorů.

V tabulce 1 jsou uvedeny časy pro obě verze implementací, jak byly popsány v kapitole 3.3. Algoritmy označeny „v1“ jsou ty, kde je realizováno odečítání vektorů knihovní funkcí PETSc. Označení „v2“ pak mají ty, kde se odečítání provádí přes pole.

Při původním testování algoritmů na obyčejném osobním notebooku byla u první verze zjištěna nadbytečná komunikace. Při přechodu na jinou architekturu superpočítače ovšem tato komunikace vymizela. Obě verze jsou proto srovnatelné vzhledem k paralelní škálovatelnosti. Ovšem implementace s knihovní funkcí jsou značně rychlejší při sekvenčním výpočtu na jednom jádře tak i při výpočtech na větším počtu jader. Proto v dalších tabulkách budou již uvedeny pouze výsledky výpočtů první verze.

Počet jader	1	25	50	100	300
Preproc. - řídké m.	6.587e+00	4.167e-01	2.210e-01	1.220e-01	4.167e-02
Preproc. - husté m.	9.354e-02	1.548e+01	7.089e+00	1.432e+00	4.069e-01
MGS v1	1.144e+01	4.687e+00	6.426e+00	5.815e+00	9.386e+00
CGS v1	1.748e+01	3.293e+00	1.541e+00	5.070e-01	1.850e-01
ICGS v1	2.335e+01	4.378e+00	2.069e+00	6.900e-01	2.680e-01
MGS v2	8.268e+01	8.645e+00	8.832e+00	6.876e+00	9.514e+00
CGS v2	8.354e+01	7.485e+00	3.794e+00	1.827e+00	6.560e-01
ICGS v2	1.112e+02	9.991e+00	5.083e+00	2.439e+00	8.980e-01
SLePc	2.329e+01	4.396e+00	2.071e+00	6.680e-01	2.180e-01
Postproc. - řídké m.	1.235e+00	1.237e-01	7.233e-02	3.967e-02	1.500e-02
Postproc. - husté m.	1.205e+00	1.183e-01	7.121e-02	3.542e-02	1.435e-02

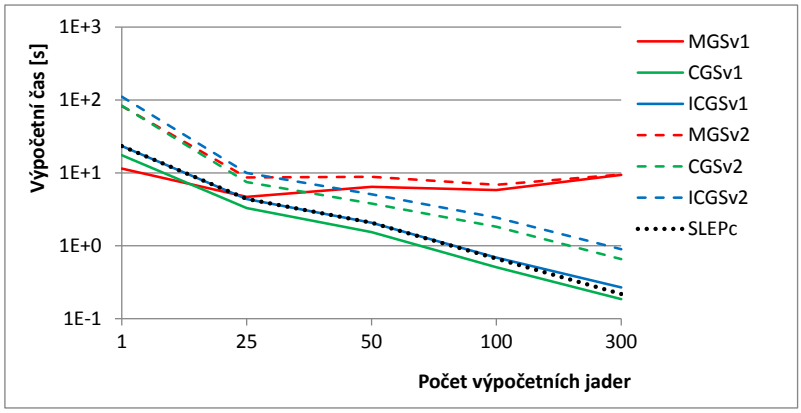
Tabulka 1: Paralelní škálovatelnost algoritmů pro matici typu 22800×900

Paralelní škálovatelnost jednotlivých algoritmů lze nejlépe porovnat při pohledu na obr. 8. Na grafu můžeme vidět, že podle předpokladů má algoritmus MGS nejhorší

vlastnosti týkající se paralelizace, které jsou způsobeny množstvím komunikace. Zatímco při sekvenčním výpočtu na jednom jádře je algoritmus MGS nejrychlejší, při spuštění výpočtu na 25 jádrech již mírně zaostává i za algoritmem ICGS, při kterém probíhá mnohem více operací v plovoucí řádové čárce. Při dalším nárůstu výpočetních jader čas algoritmu MGS dokonce mírně narůstá.

Algoritmy CGS a ICGS jsou naopak velice dobře škálovatelné. Efektivita paralelizace je u obou algoritmů srovnatelná. Algoritmus ICGS je nepatrně pomalejší z důvodu většího počtu operací, ke kterému dochází při iteračním zpřesnění.

Při porovnání algoritmů s knihovní funkcí SLEPc pro ortogonalizaci matice zjistíme, že se chová velice podobně jako implementace algoritmu ICGS. Tato funkce totiž využívá pro ortogonalizaci také klasický Gramův-Schmidtův algoritmus s iteračním zpřesněním. Funkce knihovny SLEPc je ovšem pro 300 jader nepatrně efektivnější.

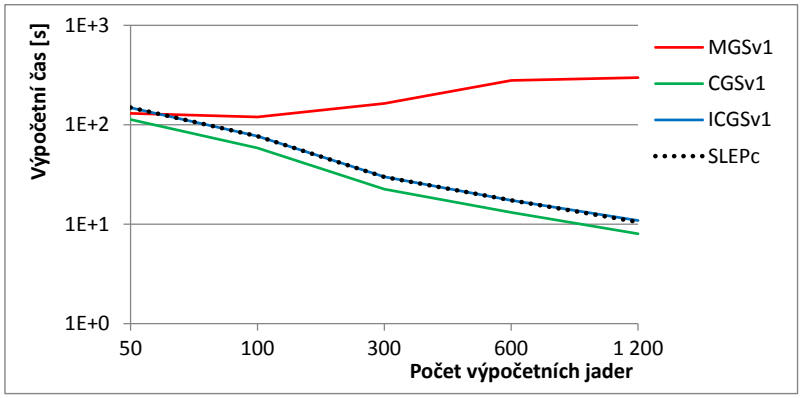


Obrázek 8: Paralelní škálovatelnost algoritmů pro matici typu 22800x900

Velmi podobné výsledky pak můžeme pozorovat i pro matice jiných rozměrů. V tabulce 2 můžeme vidět časové údaje pro matici rozměrů 93600×3600 a v tabulce 3 pak pro matici typu 106200×8100. Údaje vynesené do grafu jsou pak na obrázku 9, resp. 10.

Počet jader	50	100	300	600	1200
Preproc. - řídké m.	3.597e+00	1.743e+00	6.018e-01	3.343e-01	1.825e-01
Preproc. - husté m.	4.817e+02	2.383e+02	5.638e+01	1.915e+01	7.667e+01
MGS v1	1.301e+02	1.196e+02	1.635e+02	2.792e+02	2.978e+02
CGS v1	1.128e+02	5.832e+01	2.252e+01	1.311e+01	8.007e+00
ICGS v1	1.484e+02	7.677e+01	2.995e+01	1.736e+01	1.087e+01
SLePc	1.490e+02	7.675e+01	2.980e+01	1.740e+01	1.051e+01
Postproc. - řídké m.	1.682e+00	5.938e-01	2.123e-01	1.178e-01	7.125e-02
Postproc. - husté m.	1.620e+00	6.233e-01	1.949e-01	1.092e-01	6.823e-02

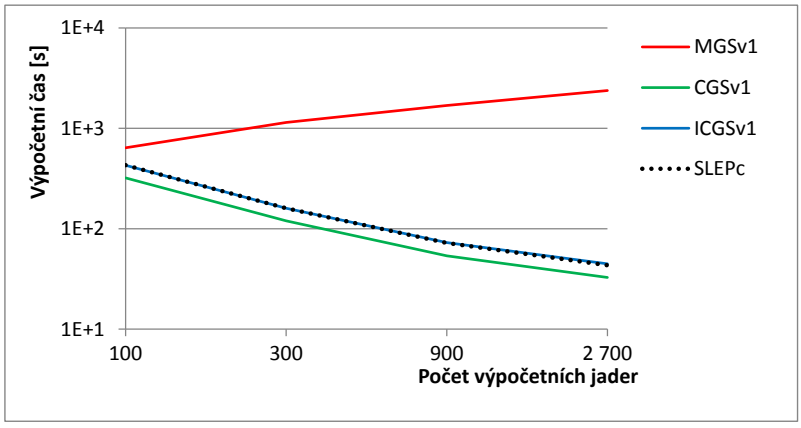
Tabulka 2: Paralelní škálovatelnost algoritmů pro matici typu 93600×3600



Obrázek 9: Paralelní škálovatelnost algoritmů pro matici typu 93600×3600

Počet jader	100	300	900	2700
Preproc. - řídké m.	4,451E+00	1,664E+00	6,385E-01	2,918E-01
MGS v1	6,411E+02	1,144E+03	1,690E+03	2,379E+03
CGS v1	3,210E+02	1,197E+02	5,373E+01	3,267E+01
ICGS v1	4,289E+02	1,600E+02	7,273E+01	4,468E+01
SLePc	4,295E+02	1,599E+02	7,204E+01	4,330E+01
Postproc. - řídké m.	1,972E+00	5,900E-01	2,618E-01	1,643E-01

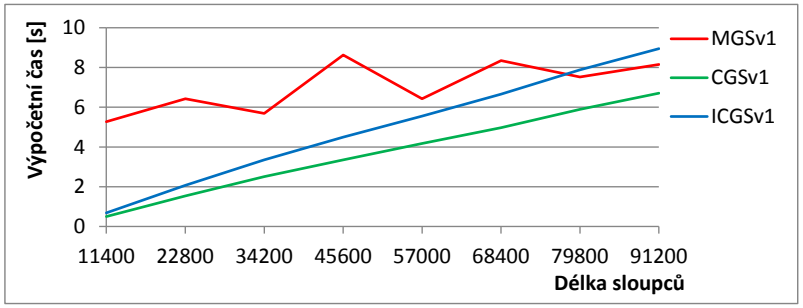
Tabulka 3: Paralelní škálovatelnost algoritmů pro matici typu 106200×8100



Obrázek 10: Paralelní škálovatelnost algoritmů pro matici typu 106200×8100

Obrázek 11 ukazuje jak se vyvíjí výpočetní čas při ortogonalizaci matice při zvětšování délky sloupců. Tato testovaná matice měla 900 sloupců a ortogonalizace byla prováděna

na 50 jádrech. Z grafu můžeme vidět, že u delších sloupců dochází k zmenšování rozdílů mezi algoritmem MGS a algoritmy CGS a ICGS. To je způsobeno zmenšením podílu komunikace na celkovém výpočetním čase. Při délce sloupců 79800 je již algoritmus MGS rychlejší než algoritmus ICGS. Lokální délka sloupce je v tomto případě 1596. V případě ortogonalizace dlouhých vektorů na menším počtu jader tak může být výhodnější algoritmus MGS než ICGS.



Obrázek 11: Závislost výpočetního času na délce ortogonalizovaných sloupců

Zbývá porovnat numerickou stabilitu jednotlivých algoritmů. Jedním ze způsobů jak určit chybu ortogonalizace je přes Frobeniovu normu matice. Chyba je pak dána dána $\left\|Q^TQ - I\right\|_F$. Jelikož násobení matic je vylice výpočetně náročná operace, byla přesnost ortogonalizace určena jiným způsobem. Nejprve byl vytvořen náhodný vektor a. Následoval výpočet $a' = Q^T(Q \cdot a)$. Požadovaná chyba pak byla porovnána s výrazem $\frac{\|a'-a\|_\infty}{\|a\|_\infty}$.

Srovnání stability výpočtu lze vidět v tabulce 4. Jednoznačně nejméně přesný je algoritmus CGS, který trpí na zaokrouhlovací chyby. Již u nejmenší uvedené testované matice je ortogonalita zcela ztracena. Naopak výpočet algoritmu MGS proběhl s poměrně malou chybou a může být použit pro ortogonalizaci podobně velkých matic. Nejlepších výsledků podle očekávání dosáhl algoritmus ICGS díky iteračnímu zpřesnění. Stejnou chybu vykazuje také ortogonalizace funkcí SLEPc.

Rozměry matice	22800×900	46800×3600	106200×8100
MGS	1e-11	1e-10	1e-10
CGS	1e+02	1e+03	1e+04
ICGS	1e-13	1e-13	1e-12
SLePc	1e-13	1e-13	1e-12

Tabulka 4: Přibližné chyby ortogonality pro různé rozměry matice

4.5 Celkové zhodnocení

Obecně nejvýhodnějším algoritmem pro ortogonalizaci se jeví ICGS. Díky iteračnímu zpřesnění má tento algoritmus jednoznačně nejlepší numerickou stabilitu. Přestože při ortogonalizaci algoritmem ICGS je prováděno více operací v plovoucí řádové čárce než u algoritmů MGS, je díky mnohem lepší paralelní škálovatelnosti při použití většího počtu jader rychlejší než MGS.

Z hlediska přesnosti výpočtu dobře obstál i algoritmus MGS. Jeho použití může být výhodné nejen při sekvenčním výpočtu, ale také v případě, kdy délka sloupců matice je velká v porovnáním s počtem použitých výpočetních jader. Časy výpočtu MGS a ICGS u zmíněných testů se srovnávaly v případě, kdy lokální délka vektoru na každém jádře byla řádově kolem 10^3.

Použití algoritmu CGS se nezdá pro větší matice výhodné, přestože se jedná o algoritmus s nejlepšími paralelními vlastnostmi. Přesnost výpočtu byla u všech uskutečněných testů velice malá.

5 Závěr

Cílem této práce bylo seznámit se s různými metodami ortogonalizace matice a následně analyzovat různé algoritmy z hlediska vhodnosti k paralelizaci.

V této práci jsem nejdříve popsal různé způsoby ortogonalizace jako je Gramův-Schmidtův ortogonalizační proces, Householderova transformace nebo Givensova rotace. Následně jsem provedl paralelizaci modifikovaného, klasického a iterovaného klasického Gramova-Schmidtova algoritmu. V rámci práce jsem se seznámil s knihovnou pro paralelizaci vědeckých výpočtů PETSc. Implementace tří zmíněných algoritmů byla provedena právě pomocí této knihovny a byla v textu této práce podrobně popsána.

Vzniklé implementace byly testovány na maticích různých rozměrů vzniklých při aplikování metody TFETI na úlohu ocelového nosníku, kde ortogonalizace skutečně může významně pomoci při redukci výpočetní náročnosti. Paralelní škálovatelnost byla otestována na superpočítači HECToR a výpočet byl spuštěn u některých matic až na 2700 jádrech. Testy ukázaly, že obecně je nejvýhodnějším algoritmem pro ortogonalizaci ICGS z důvodu velmi špatné stability výpočtu algoritmem CGS a slabé paralelní škálovatelnosti algoritmu MGS.

Vlastní PETSc implementace by dále mohly být vylepšeny především v oblasti efektivity výběru sloupců z matice, která je dosud realizována pomocí pomalé knihovní funkce, a to zvláště u hustých matic. Dále by bylo možné se zabývat paralelizací i jiných metod pro ortogonalizaci matice.

6 Reference

[1] Hapla, V., Horák, D., *TFETI coarse space projectors parallelization strategies,* In: Parallel processing and applied mathematics: 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised selected papers, part I. New York: Springer, 2012, pp. 152–162. LNCS, 7203. ISBN 978-364231463-6, ISSN 03029

[2] Lingen, F., J., *Efficient Gram-Schmidt orthonormalisation on parallel computers.* Research report, Dep. of Aerospace Engineering, Delft University of Technology, 1999.

[3] Kozubek, T., Brzobohatý, T., Hapla, V., Jarošová, M., Markopoulos, A., *Lineární algebra s Matlabem,* Skriptum VŠB-Technické Univerzity Ostrava, 2012.

[4] Balay S., Brown J., Buschelman K., Eijkhout V., Gropp W., Kaushik D., Knepley M., Curfman McInnes L., Smith B. Zhang H., *PETSc 3.3 Users Manual*, http://www.mcs.anl.gov/petsc/, Argonne National Laboratory, 2012.

[5] Campos C., Román J. E., Romero E., Tomás A., *SLEPc Users Manual 3.3*, http://www.grycap.upv.es/slepc, Universitat Politécnica de Valencia, 2012.

[6] Kotas P., *Efektivní SVD a jeho využití při zpracování biometrických dat.* Diplomová práce, Fakulta elektrotechniky a informatiky, VŠB-Technická Univerzita Ostrava, 2009. Vedoucí diplomové práce Doc. Mgr. Vít Vondrák, Ph.D.

[7] Jirůtková P., *Paralelizace řešení eliptických okrajových úloh pomocí TFETI metody rozložení oblastí.* Bakalářská práce, VŠB-Technická Univerzita Ostrava, Fakulta elektrotechniky a informatiky, 2012. Vedoucí bakalářské práce Doc. Ing. Tomáš Kozubek, Ph.D.

[8] *HECToR.* [online] [cit. 2013-04-24]. Dostupné z http://www.hector.ac.uk/

A Výpisy výkonnosti implementací

Zde jsou umístěny výpisy jednotlivých algoritmů, které zobrazují množství operací, komunikace nebo paměťových nároků. Pro každý z algoritmů jsou uvedené dvě verze implementace. V první verzi je odečítání vektorů prováděno pomocí funkce PETSc, kde ovšem dochází k přebytečné komunikaci mezi procesy. V druhé verzi implementace je odečítání implementováno pomocí pole a přebytečná komunikace je tak odstraněna.

Přiložené výpisy o výkonosti jednotlivých implementací byly získány při ortogonalizace matice typu 1680×144. Výpočet probíhal na notebooku na čtyřech jádrech.

```
---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

                         Max       Max/Min        Avg      Total
Time (sec):           3.450e+00      1.00020   3.450e+00
Objects:              2.250e+02      1.00000   2.250e+02
Flops:                1.510e+07      1.00000   1.510e+07  6.038e+07
Flops/sec:            4.376e+06      1.00020   4.375e+06  1.750e+07
Memory:               2.027e+06      1.00987              8.068e+06
MPI Messages:         1.800e+01      1.50000   1.350e+01  5.400e+01
MPI Message Lengths:  7.944e+03      1.30187   5.193e+02  2.804e+04
MPI Reductions:       2.126e+04      1.00000

------------------------------------------------------------------------------------------------------------------------
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %f %M %L %R  %T %f %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
MatMult               26 1.0 3.2693e-02 1.0 3.14e+06 1.0 0.0e+00 0.0e+00 5.2e+01  1 21  0 53  0   1 21  0 53  0   384
MatMultTranspose      13 1.0 1.8707e-02 1.1 1.57e+06 1.0 0.0e+00 0.0e+00 3.9e+01  1 10  0 53  0   1 10  0 53  0   336
MatAssemblyBegin       9 1.0 1.5254e-02 2.9 0.00e+00 0.0 0.0e+00 0.0e+00 1.2e+01  0  0  0  0  0   0  0  0  0  0     0
MatAssemblyEnd         9 1.0 1.6734e-02 1.2 0.00e+00 0.0 5.4e+01 2.4e+02 9.2e+01  0  0100 47  0   0  0100 47  0     0
MatGetRow          60516 1.0 8.6399e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  2  0  0  0  0   2  0  0  0  0     0
MatTrnMatMult          1 1.0 1.8278e-02 1.2 5.04e+05 1.0 1.8e+01 3.4e+02 6.2e+01  1  3 33 22  0   1  3 33 22  0   110
MatGetLocalMat         2 1.0 1.0370e-03 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecTDot            10296 1.0 1.5264e+00 1.0 8.64e+06 1.0 0.0e+00 0.0e+00 1.0e+04 44 57  0  0 48  44 57  0  0 48    23
VecNorm              170 1.0 2.1841e-02 1.0 1.21e+05 1.0 0.0e+00 0.0e+00 1.7e+02  1  1  0  0  1   1  1  0  0  1    22
VecScale             157 1.0 1.2092e-03 1.1 6.05e+04 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0   200
VecCopy               13 1.0 4.7734e-05 1.6 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY            10309 1.0 4.3991e-02 1.1 2.63e+06 1.0 0.0e+00 0.0e+00 0.0e+00  1 17  0  0  0   1 17  0  0  0   240
VecAssemblyBegin       3 1.0 3.7169e-03 3.0 0.00e+00 0.0 0.0e+00 0.0e+00 9.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAssemblyEnd         3 1.0 1.0072e-05 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecScatterBegin       29 1.0 6.1458e-03 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 4.2e+01  0  0  0 53  0   0  0  0 53  0     0
Ortogonalizace         1 1.0 3.0427e+00 1.0 1.15e+07 1.0 0.0e+00 0.0e+00 2.1e+04 88 76  0  0 98  88 76  0  0 98    15
------------------------------------------------------------------------------------------------------------------------
Memory usage is given in bytes:
Object Type          Creations   Destructions     Memory  Descendants' Mem.
Reports information only for process 0.
              Matrix    22              8       773864     0
              Vector   177            155       661892     0
      Vector Scatter     8              5         2084     0
           Index Set    13             13         7764     0
         PetscRandom     4              0            0     0
              Viewer     1              0            0     0
```

Obrázek 12: Výpis údajů pro ortogonalizaci algoritmem MGS s odečítáním pomoci PETSc funkce

```
---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

                       Max       Max/Min        Avg      Total
Time (sec):           1.958e+00      1.00057   1.957e+00
Objects:              2.250e+02      1.00000   2.250e+02
Flops:                1.246e+07      1.00000   1.246e+07  4.985e+07
Flops/sec:            6.369e+06      1.00057   6.367e+06  2.547e+07
Memory:               2.027e+06      1.00987              8.068e+06
MPI Messages:         1.800e+01      1.50000   1.350e+01  5.400e+01
MPI Message Lengths:  7.944e+03      1.30187   5.193e+02  2.804e+04
MPI Reductions:       1.096e+04      1.00000

------------------------------------------------------------------------------------------------------------------------
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %f %M %L %R  %T %f %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
MatMult               26 1.0 3.0786e-02 1.0 3.14e+06 1.0 0.0e+00 0.0e+00 5.2e+01  2 25  0 53  0   2 25  0 53  0   408
MatMultTranspose      13 1.0 1.6240e-02 1.0 1.57e+06 1.0 0.0e+00 0.0e+00 3.9e+01  1 13  0 53  0   1 13  0 53  0   387
MatAssemblyBegin       9 1.0 1.4562e-02 2.0 0.00e+00 0.0 0.0e+00 0.0e+00 1.2e+01  1  0  0  0  0   1  0  0  0  0     0
MatAssemblyEnd         9 1.0 2.0988e-02 1.3 0.00e+00 0.0 5.4e+01 2.4e+02 9.2e+01  1  0100 47  1   1  0100 47  1     0
MatGetRow          60516 1.0 9.4633e-02 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  5  0  0  0  0   5  0  0  0  0     0
MatTrnMatMult          1 1.0 2.4103e-02 1.0 5.04e+05 1.0 1.8e+01 3.4e+02 6.2e+01  1  4 33 22  1   1  4 33 22  1    84
MatGetLocalMat         2 1.0 1.0620e-03 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecTDot            10296 1.0 1.3730e+00 1.0 8.64e+06 1.0 0.0e+00 0.0e+00 1.0e+04 69 69  0  0 94  69 69  0  0 94    25
VecNorm              170 1.0 2.5072e-02 1.1 1.21e+05 1.0 0.0e+00 0.0e+00 1.7e+02  1  1  0  0  2   1  1  0  0  2    19
VecCopy               13 1.0 5.1399e-05 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY               13 1.0 6.1917e-05 2.2 9.36e+02 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0    60
VecAssemblyBegin       3 1.0 4.1619e-03 2.7 0.00e+00 0.0 0.0e+00 0.0e+00 9.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAssemblyEnd         3 1.0 1.7114e-05 1.9 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecScatterBegin       29 1.0 2.7702e-03 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 4.2e+01  0  0  0 53  0   0  0  0 53  0     0
VecSetRandom          13 1.0 1.4671e-04 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
Ortogonalizace         1 1.0 1.4937e+00 1.0 8.82e+06 1.0 0.0e+00 0.0e+00 1.0e+04 76 71  0  0 95  76 71  0  0 95    24
------------------------------------------------------------------------------------------------------------------------

Memory usage is given in bytes:
Object Type          Creations   Destructions     Memory  Descendants' Mem.
Reports information only for process 0.
              Matrix    22              8       773864     0
              Vector   177            155       661892     0
      Vector Scatter     8              5         2084     0
           Index Set    13             13         7764     0
         PetscRandom     4              0            0     0
              Viewer     1              0            0     0
```

Obrázek 13: Výpis údajů pro ortogonalizaci algoritmem MGS s odečítáním pomocí polí

```
---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

                         Max       Max/Min        Avg      Total
Time (sec):           2.207e+00      1.00013   2.207e+00
Objects:              2.900e+02      1.00000   2.900e+02
Flops:                2.425e+07      1.00000   2.425e+07  9.700e+07
Flops/sec:            1.099e+07      1.00013   1.099e+07  4.395e+07
Memory:               2.027e+06      1.00987              8.068e+06
MPI Messages:         1.800e+01      1.50000   1.350e+01  5.400e+01
MPI Message Lengths:  1.169e+04      1.18708   7.967e+02  4.302e+04
MPI Reductions:       1.129e+04      1.00000

------------------------------------------------------------------------------------------------------------------------
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %f %M %L %R  %T %f %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
MatMult               52 1.0 6.7884e-02 1.0 6.28e+06 1.0 0.0e+00 0.0e+00 1.0e+02  3 26  0 70  1   3 26  0 70  1   370
MatMultTranspose      26 1.0 3.5857e-02 1.0 3.14e+06 1.0 0.0e+00 0.0e+00 7.8e+01  2 13  0 70  1   2 13  0 70  1   350
MatAssemblyBegin       9 1.0 9.0572e-03 3.3 0.00e+00 0.0 0.0e+00 0.0e+00 1.2e+01  0  0  0  0  0   0  0  0  0  0     0
MatAssemblyEnd         9 1.0 2.0683e-02 1.2 0.00e+00 0.0 5.4e+01 2.4e+02 9.2e+01  1  0100 30  1   1  0100 30  1     0
MatGetRow          60516 1.0 8.8744e-02 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  4  0  0  0  0   4  0  0  0  0     0
MatTrnMatMult          1 1.0 2.1676e-02 1.2 5.04e+05 1.0 1.8e+01 3.4e+02 6.2e+01  1  2 33 14  1   1  2 33 14  1    93
MatGetLocalMat         2 1.0 1.4089e-03 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecMDot              143 1.0 7.7769e-02 1.1 8.64e+06 1.0 0.0e+00 0.0e+00 1.4e+02  3 36  0  0  1   3 36  0  0  1   444
VecNorm              196 1.0 3.8949e-02 1.1 1.21e+05 1.0 0.0e+00 0.0e+00 2.0e+02  2  0  0  0  2   2  0  0  0  2    12
VecAXPY               15 1.0 6.6278e-05 2.1 1.08e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0    65
VecMAXPY             143 1.0 6.2435e-02 1.0 8.65e+06 1.0 0.0e+00 0.0e+00 0.0e+00  3 36  0  0  0   3 36  0  0  0   554
VecAssemblyBegin       3 1.0 3.1961e-03 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 9.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAssemblyEnd         3 1.0 1.0054e-05 1.3 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecScatterBegin       55 1.0 8.6379e-03 1.3 0.00e+00 0.0 0.0e+00 0.0e+00 8.1e+01  0  0  0 70  1   0  0  0 70  1     0
Ortogonalizace         1 1.0 1.7036e+00 1.0 1.75e+07 1.0 0.0e+00 0.0e+00 1.1e+04 77 72  0  0 94  77 72  0  0 94    41
------------------------------------------------------------------------------------------------------------------------
Memory usage is given in bytes:
Object Type          Creations   Destructions     Memory  Descendants' Mem.
Reports information only for process 0.
              Matrix    35              8       773864     0
              Vector   216            155       661892     0
      Vector Scatter     8              5         2084     0
           Index Set    13             13         7764     0
         PetscRandom    17              0            0     0
              Viewer     1              0            0     0
```

Obrázek 14: Výpis údajů pro ortogonalizaci algoritmem CGS s odečítáním pomoci PETSc funkce

```
---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

                         Max       Max/Min        Avg      Total
Time (sec):           6.702e-01      1.00543   6.691e-01
Objects:              2.900e+02      1.00000   2.900e+02
Flops:                1.560e+07      1.00000   1.560e+07  6.240e+07
Flops/sec:            2.340e+07      1.00543   2.331e+07  9.326e+07
Memory:               2.027e+06      1.00987              8.068e+06
MPI Messages:         1.800e+01      1.50000   1.350e+01  5.400e+01
MPI Message Lengths:  1.169e+04      1.18708   7.967e+02  4.302e+04
MPI Reductions:       9.920e+02      1.00000

------------------------------------------------------------------------------------------------------------------------
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %f %M %L %R  %T %f %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
MatMult               52 1.0 6.6648e-02 1.0 6.28e+06 1.0 0.0e+00 0.0e+00 1.0e+02 10 40  0 70 10  10 40  0 70 10   377
MatMultTranspose      26 1.0 3.7052e-02 1.1 3.14e+06 1.0 0.0e+00 0.0e+00 7.8e+01  5 20  0 70  8   5 20  0 70  8   339
MatAssemblyBegin       9 1.0 1.9719e-02 2.9 0.00e+00 0.0 0.0e+00 0.0e+00 1.2e+01  2  0  0  0  1   2  0  0  0  1     0
MatAssemblyEnd         9 1.0 2.2787e-02 1.1 0.00e+00 0.0 5.4e+01 2.4e+02 9.2e+01  3  0100 30  9   3  0100 30  9     0
MatGetRow          60516 1.0 9.6387e-02 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00 14  0  0  0  0  14  0  0  0  0     0
MatTrnMatMult          1 1.0 1.7764e-02 1.0 5.04e+05 1.0 1.8e+01 3.4e+02 6.2e+01  3  3 33 14  6   3  3 33 14  6   113
MatGetLocalMat         2 1.0 1.1029e-03 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecMDot              143 1.0 4.7205e-02 1.2 8.64e+06 1.0 0.0e+00 0.0e+00 1.4e+02  6 55  0  0 14   6 55  0  0 14   732
VecNorm              196 1.0 3.9388e-02 1.8 1.21e+05 1.0 0.0e+00 0.0e+00 2.0e+02  5  1  0  0 20   5  1  0  0 20    12
VecScale             170 1.0 7.5907e-04 1.1 6.05e+04 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0   319
VecCopy               26 1.0 9.9054e-05 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY               15 1.0 7.1394e-05 2.3 1.08e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0    61
VecAssemblyBegin       3 1.0 7.9097e-04 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 9.0e+00  0  0  0  0  1   0  0  0  0  1     0
VecAssemblyEnd         3 1.0 1.1120e-05 1.6 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecScatterBegin       55 1.0 9.2035e-03 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 8.1e+01  1  0  0 70  8   1  0  0 70  8     0
Ortogonalizace         1 1.0 1.3681e-01 1.0 8.82e+06 1.0 0.0e+00 0.0e+00 2.9e+02 20 57  0  0 29  20 57  0  0 29   258
------------------------------------------------------------------------------------------------------------------------
Memory usage is given in bytes:
Object Type          Creations   Destructions     Memory  Descendants' Mem.
Reports information only for process 0.
              Matrix    35              8       773864     0
              Vector   216            155       661892     0
      Vector Scatter     8              5         2084     0
           Index Set    13             13         7764     0
         PetscRandom    17              0            0     0
              Viewer     1              0            0     0
```

Obrázek 15: Výpis údajů pro ortogonalizaci algoritmem CGS s odečítáním pomocí polí

```
---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

                         Max       Max/Min        Avg      Total
Time (sec):           2.554e+00      1.00052   2.553e+00
Objects:              2.150e+02      1.00000   2.150e+02
Flops:                2.686e+07      1.00000   2.686e+07  1.074e+08
Flops/sec:            1.052e+07      1.00052   1.052e+07  4.207e+07
Memory:               2.027e+06      1.00987              8.068e+06
MPI Messages:         1.800e+01      1.50000   1.350e+01  5.400e+01
MPI Message Lengths:  7.368e+03      1.33333   4.767e+02  2.574e+04
MPI Reductions:       1.493e+04      1.00000

------------------------------------------------------------------------------------------------------------------------
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %f %M %L %R  %T %f %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
MatMult              22 1.0 3.1626e-02 1.0 2.65e+06 1.0 0.0e+00 0.0e+00 4.4e+01  1 10  0 49  0   1 10  0 49  0   336
MatMultTranspose     11 1.0 1.8352e-02 1.1 1.33e+06 1.0 0.0e+00 0.0e+00 3.3e+01  1  5  0 49  0   1  5  0 49  0   290
MatAssemblyBegin      9 1.0 1.4578e-02 4.4 0.00e+00 0.0 0.0e+00 0.0e+00 1.2e+01  0  0  0  0  0   0  0  0  0  0     0
MatAssemblyEnd        9 1.0 2.2276e-02 1.4 0.00e+00 0.0 5.4e+01 2.4e+02 9.2e+01  1  0100 51  1   1  0100 51  1     0
MatGetRow         60516 1.0 9.0657e-02 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  3  0  0  0  0   3  0  0  0  0     0
MatTrnMatMult         1 1.0 1.5602e-02 1.2 5.04e+05 1.0 1.8e+01 3.4e+02 6.2e+01  1  2 33 23  0   1  2 33 23  0   129
MatGetLocalMat        2 1.0 1.1902e-03 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecMDot             190 1.0 9.6179e-02 1.1 1.17e+07 1.0 0.0e+00 0.0e+00 1.9e+02  4 43  0  0  1   4 43  0  0  1   485
VecNorm             356 1.0 5.7370e-02 1.1 2.81e+05 1.0 0.0e+00 0.0e+00 3.6e+02  2  1  0  0  2   2  1  0  0  2    20
VecCopy              11 1.0 4.7174e-05 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecSet              166 1.0 7.4709e-04 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY              11 1.0 5.0056e-05 2.0 7.92e+02 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0    63
VecMAXPY            190 1.0 7.9730e-02 1.0 1.17e+07 1.0 0.0e+00 0.0e+00 0.0e+00  3 44  0  0  0   3 44  0  0  0   586
VecAssemblyBegin      3 1.0 5.1291e-03 3.1 0.00e+00 0.0 0.0e+00 0.0e+00 9.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAssemblyEnd        3 1.0 1.5173e-05 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecScatterBegin      25 1.0 8.7856e-03 2.1 0.00e+00 0.0 0.0e+00 0.0e+00 3.6e+01  0  0  0 49  0   0  0  0 49  0     0
Ortogonalizace        1 1.0 2.1288e+00 1.0 2.37e+07 1.0 0.0e+00 0.0e+00 1.4e+04 83 88  0  0 97  83 88  0  0 97    45
------------------------------------------------------------------------------------------------------------------------

Memory usage is given in bytes:

Object Type          Creations   Destructions    Memory  Descendants' Mem.
Reports information only for process 0.
              Matrix    20              8       773864     0
              Vector   171            155       661892     0
      Vector Scatter     8              5         2084     0
           Index Set    13             13         7764     0
         PetscRandom     2              0            0     0
              Viewer     1              0            0     0
```

Obrázek 16: Výpis údajů pro ortogonalizaci algoritmem ICGS s odečítáním pomoci PETSc funkce

```
---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

                         Max       Max/Min        Avg      Total
Time (sec):           6.004e-01      1.00006   6.004e-01
Objects:              2.150e+02      1.00000   2.150e+02
Flops:                1.517e+07      1.00000   1.517e+07  6.069e+07
Flops/sec:            2.527e+07      1.00006   2.527e+07  1.011e+08
Memory:               2.027e+06      1.00987              8.068e+06
MPI Messages:         1.800e+01      1.50000   1.350e+01  5.400e+01
MPI Message Lengths:  7.368e+03      1.33333   4.767e+02  2.574e+04
MPI Reductions:       1.015e+03      1.00000

------------------------------------------------------------------------------------------------------------------------
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %f %M %L %R  %T %f %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
MatMult              22 1.0 2.8608e-02 1.0 2.65e+06 1.0 0.0e+00 0.0e+00 4.4e+01  5 17  0 49  4   5 17  0 49  4   371
MatMultTranspose     11 1.0 1.6908e-02 1.1 1.33e+06 1.0 0.0e+00 0.0e+00 3.3e+01  3  9  0 49  3   3  9  0 49  3   314
MatAssemblyBegin      9 1.0 8.6609e-03 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 1.2e+01  1  0  0  0  1   1  0  0  0  1     0
MatAssemblyEnd        9 1.0 1.8813e-02 1.1 0.00e+00 0.0 5.4e+01 2.4e+02 9.2e+01  3  0100 51  9   3  0100 51  9     0
MatGetRow         60516 1.0 8.8404e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00 15  0  0  0  0 15  0  0  0  0     0
MatTrnMatMult         1 1.0 2.2192e-02 1.2 5.04e+05 1.0 1.8e+01 3.4e+02 6.2e+01  4  3 33 23  6   4  3 33 23  6    91
MatGetLocalMat        2 1.0 1.6090e-03 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecMDot             190 1.0 5.9835e-02 1.2 1.17e+07 1.0 0.0e+00 0.0e+00 1.9e+02  9 77  0  0 19   9 77  0  0 19   780
VecNorm             356 1.0 5.5374e-02 1.5 2.81e+05 1.0 0.0e+00 0.0e+00 3.6e+02  7  2  0  0 35   7  2  0  0 35    20
VecCopy              11 1.0 4.1297e-05 1.5 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY              11 1.0 4.9033e-05 2.0 7.92e+02 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0    65
VecAssemblyBegin      3 1.0 1.0190e-03 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 9.0e+00  0  0  0  0  1   0  0  0  0  1     0
VecAssemblyEnd        3 1.0 9.9095e-06 1.7 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecScatterBegin      25 1.0 8.6989e-03 1.9 0.00e+00 0.0 0.0e+00 0.0e+00 3.6e+01  1  0  0 49  4   1  0  0 49  4     0
Ortogonalizace        1 1.0 1.8822e-01 1.0 1.20e+07 1.0 0.0e+00 0.0e+00 5.2e+02 31 79  0  0 52  31 79  0  0 52   255
------------------------------------------------------------------------------------------------------------------------
Memory usage is given in bytes:

Object Type          Creations   Destructions     Memory  Descendants' Mem.
Reports information only for process 0.
              Matrix    20              8       773864     0
              Vector   171            155       661892     0
      Vector Scatter     8              5         2084     0
           Index Set    13             13         7764     0
         PetscRandom     2              0            0     0
              Viewer     1              0            0     0
```

Obrázek 17: Výpis údajů pro ortogonalizaci algoritmem ICGS s odečítáním pomocí polí

```
---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

                         Max       Max/Min        Avg      Total
Time (sec):           2.853e+00      1.00007   2.853e+00
Objects:              2.160e+02      1.00000   2.160e+02
Flops:                2.745e+07      1.00000   2.745e+07  1.098e+08
Flops/sec:            9.621e+06      1.00007   9.621e+06  3.848e+07
Memory:               2.032e+06      1.00984              8.090e+06
MPI Messages:         1.800e+01      1.50000   1.350e+01  5.400e+01
MPI Message Lengths:  7.368e+03      1.33333   4.767e+02  2.574e+04
MPI Reductions:       1.590e+04      1.00000

------------------------------------------------------------------------------------------------------------------------
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %f %M %L %R  %T %f %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
IPOrthogonalize      144 1.0 2.4376e+00 1.0 2.42e+07 1.0 0.0e+00 0.0e+00 1.5e+04 85 88  0  0 95  85 88  0  0 95    40
IPInnerProduct       782 1.0 1.1623e-01 1.1 1.22e+07 1.0 0.0e+00 0.0e+00 2.0e+02  4 44  0  0  1   4 44  0  0  1   420
MatMult               22 1.0 2.4660e-02 1.0 2.65e+06 1.0 0.0e+00 0.0e+00 4.4e+01  1 10  0 49  0   1 10  0 49  0   431
MatMultTranspose      11 1.0 1.2975e-02 1.0 1.33e+06 1.0 0.0e+00 0.0e+00 3.3e+01  0  5  0 49  0   0  5  0 49  0   410
MatAssemblyBegin       9 1.0 7.2679e-03 1.3 0.00e+00 0.0 0.0e+00 0.0e+00 1.2e+01  0  0  0  0  0   0  0  0  0  0     0
MatAssemblyEnd         9 1.0 1.8756e-02 1.3 0.00e+00 0.0 5.4e+01 2.4e+02 9.2e+01  1  0100 51  1   1  0100 51  1     0
MatGetRow          60516 1.0 8.6984e-02 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  3  0  0  0  0   3  0  0  0  0     0
MatTrnMatMult          1 1.0 1.9087e-02 1.2 5.04e+05 1.0 1.8e+01 3.4e+02 6.2e+01  1  2 33 23  0   1  2 33 23  0   106
MatGetLocalMat         2 1.0 1.0491e-03 1.2 0.00e+00 0.0 0.0e+00 0.0e+00 4.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecNorm               22 1.0 1.7912e-03 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 2.2e+01  0  0  0  0  0   0  0  0  0  0     0
VecCopy               11 1.0 3.2224e-05 1.3 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY               11 1.0 4.6476e-05 2.0 7.92e+02 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0    68
VecMAXPY             195 1.0 9.2812e-02 1.0 1.20e+07 1.0 0.0e+00 0.0e+00 0.0e+00  3 44  0  0  0   3 44  0  0  0   519
VecAssemblyBegin       3 1.0 3.7429e-03 2.9 0.00e+00 0.0 0.0e+00 0.0e+00 9.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAssemblyEnd         3 1.0 1.0961e-05 1.6 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecScatterBegin       25 1.0 4.1802e-03 1.4 0.00e+00 0.0 0.0e+00 0.0e+00 3.6e+01  0  0  0 49  0   0  0  0 49  0     0
VecReduceArith       391 1.0 6.7538e-02 1.0 1.22e+07 1.0 0.0e+00 0.0e+00 0.0e+00  2 44  0  0  0   2 44  0  0  0   722
VecReduceComm        196 1.0 4.3008e-02 1.1 0.00e+00 0.0 0.0e+00 0.0e+00 2.0e+02  1  0  0  0  1   1  0  0  0  1     0
Ortogonalizace         1 1.0 2.4874e+00 1.0 2.43e+07 1.0 0.0e+00 0.0e+00 1.5e+04 87 88  0  0 97  87 88  0  0 97    39
------------------------------------------------------------------------------------------------------------------------

Memory usage is given in bytes:
Object Type          Creations   Destructions     Memory  Descendants' Mem.
Reports information only for process 0.
       Inner product     1              1          364     0
              Matrix    20              8       773864     0
              Vector   171            156       666112     0
      Vector Scatter     8              5         2084     0
           Index Set    13             13         7764     0
         PetscRandom     2              0            0     0
              Viewer     1              0            0     0
```

Obrázek 18: Výpis údajů pro ortogonalizaci SLEPc funkcí

B Obsah přiloženého CD

- Bakalářská práce v elektronické podobě
- Zdrojové kódy PETSc implementací ortogonalizačních algoritmů, generátoru matic aj.
- Naměřené hodnoty při numerických experimentech